DETERMINING THE POSSIBILITY OF ADVERSE EFFECTS ARISING FROM A CODE CHANGE

Field of the invention

The present invention relates to the field of software development, and to large scale computer programs exhibiting complexity. It relates particularly to determining the possibility of adverse effects in other components of the program arising from code changes elsewhere in the program.

Background

Any large software product/program is most usually developed by large, numerous and possibly geographically distributed teams of programmers. This presents several challenges. One of the challenges is to ensure that code changes introduced in one component (or part) do not affect the correct execution of other dependent components (or parts). The dependency of such components can be due to referencing a type, or due to consuming data produced by that type. Typically, the dependency between the various components is not known accurately, due to incomplete specifications or due to the specification not being up-to-date.

One approach to this problem is manually trying to identify adverse affects (leading to errors), but this is quite impractical for complex software.

US Patent No. 5,694,540, issued to Humelsine et al on December 2, 1997, teaches a set of tests to run on a computer program as a regression test that provides an approximation to the level of testing that is achieved by full regression. A modification request is associated with a test case and the files that change due to the modification are recorded. The test cases associated with the files that are modified by the modification are run.

US Patent Publication No. 2003/0018950A1, in the name Sparks et al, published on January 23, 2003, describes an approach where classes are dynamically reloaded if a code change is detected. A developer can see the result of a change after a build/package step.

These known methods provide only a partial solution to predicting adverse effects. There thus remains a need for an automated approach to more completely detecting adverse effects in other program components resulting from code changes.

Summary

Some of the set of classes within a computer program are important in the sense that the most useful information about the software can be derived from these classes alone. The important classes within the software are identified, as well as any dependent classes. Test cases are defined and associated with all classes. A code change for a class invokes the relevant test case or cases being run. The corresponding test case or cases for any dependent class are also run. If they run successfully (in the sense that the expected results arise), then it is highly likely that the changes introduced in the first class are not affecting the correct execution of the dependent classes.

The key, therefore, is the ability to suggest which classes would get affected if the behavior of a given class is changed. A class can potentially get affected if it depends on a class that is changed. The dependency can be because it directly refers to the class being changed, or indirectly because it consumes data that is generated or modified by the other class.

Description of drawings

Fig. 1 is a diagram illustrating class dependencies.

Fig. 2 is a flow diagram of an initialization process.

Fig. 3 is a flow diagram of determining the possibility of adverse effects from code changes.

Fig. 4 shows the inheritance structure of a set of classes.

Fig. 5 is a schematic representation of a computer system suitable for performing the techniques described with reference to **Figs. 2** to **4**.

Detailed description

Definition of terms

It is useful to introduce a few terms:

Class: any type (e.g., classes and interfaces in Java™) in an object oriented programing language is a "class".

Test Case: Test cases are used to verify the correctness of the software. These can be of various types, e.g., "unit test, "functional test", "system test". These are collectively called "test cases".

Dependency: Consider the example given in **Fig. 1**. Class A has a direct dependency on class B, as A refers to B. Class C modifies persistent data represented by class D, which is consumed by class E in turn modified persistent data represented by class F. This data (class F) is consumed by class G. Therefore, both class E and class G have indirect dependency on class C. Any change in class C can potentially affect classes E and G.

Overview

An embodiment of the invention will be given using the example of Java™ programing language, being one type of object oriented programing languages.

The method broadly includes of the initial steps, as shown in **Fig. 2**.

The reference structure of the software is found (step **10**). Next, the important classes of the software are identified (step **12**). These important classes include the classes used for representing the persistent data (e.g., the entity bean in a J2EE environment). Next, the references to the important classes are found (step **14**) and the methods that are invoked for each of the important classes are found (step **16**).

The dependency structure of the software is now determined (step **18**), leading to identifying the directly dependent classes (step **20**) and the indirectly dependent classes (step **22**). The indirect dependencies are identified by looking for a producer/consumer relation for persistent data. The *producer* of data is a class that makes a non-read-only call (possibly in addition to some read-only calls) to the classes representing the persistent data, while the *consumer* of data is a class that makes a read-only call to the classes representing the persistent data.

The test case or cases for each class are now defined (step **24**). This involves specifying a set of steps to be performed and the expected results at each step. The authors of such test cases are skilled programmers, and the nature of the test cases depends upon the software high level specficiations. In execution, if all the steps give the expected results, then the test case is considered to be successful. The test cases are associated with the "important" and dependent classes (step **26**).

Now, with reference to **Fig. 3**, when the code for a particular class is changed (step **30**), the test case or cases associated with it are run (step **32**). The dependent (ie. both direct and indirect) classes for this type are also found (step **34**), and the associated test cases are run (step **36**). If a class is not important, then there will not be any associated test caese to be run.

If any of the test cases fail (step **38**) then appropriate action is taken (step **40**), else the process ends (step **42**). Such action can include informing the programmer, who can decide whether to retain the changes made in the code, or not.

If the developer wants to retain the changes, then the further action to take is to notify the owners of the classes for which test cases failed, including the details of change in the code triggered this failure.

Detailed implementation

Identifying the important classes

Typically, a large software program would define a template of the important classes by providing a set of classes and/or interfaces that the important business classes must extend/implement. These templates serve as the start points. For example, some of the important business classes/interfaces in the IBM WebSphere Commerce™ suite are the controller command interfaces, controller command implementation classes, task command interfaces, task command implementation classes, etc. Each controller command interface must extend a particular interface called com.ibm.commerce.command.ControllerCommand, either directly or by extending another interface, which is a controller command interface in turn.

To identify the important classes, the source (or the object) code is scanned to find the class names and their super classes (the classes this class is extending and/or implementing). A graph of the inheritance structure is then built using this information. This graph, in one form, is a directed acyclic graph **50** as shown in **Fig. 4**. Node B **52** is a direct descendent of node A **54** in this graph if node B is a super class of node A. Node D **56** is an indirect descendent of node A if node C **60** is a direct or indirect descendent of node B, where node B, in turn, is a direct or indirect descendent of node D. Node G **62** is not a descendent or super class of any other node. All the direct or indirect descendents of the start points are important classes. Beginning at each start point, the important types

are found from the graph using a depth first search or a breadth first search. For a start point of node B **52**, the important classes are class B itself, and classes D and E.

Finding references to a given class

There are a number standard utilities available to find the references to a given class. Additionally, utilities are also able to indicate which members of the given class are being accessed. By using any such utility, each method in the given class is represented by the set (possibly ordered) of member accesses of the important classes as identified above. The entire cell graph is generated, then filtered to remove all the classes that are not within the set of important classes.

A suitable utility is described in a document A Guide to the Information Added by Document Enhancer for Java, published by IBM Haifa Research Labs, Haifa, Israel, incorporated herein by reference. The utility can be downloaded from: http://www.haifa.il.ibm.com/projects/systems/ple/DEJava/index.html.

Using the reference to important classes information to find the dependencies

The direct dependencies are easily found. If there is a reference to a given class, it is a direct dependency.

To detect an indirect dependency, the classes that represent the persistent data are found. The user has to provide the template for the classes representing the persistent data in the set of start points, and indicate that these templates represent persistent data. All the classes that have these start points as their direct or indirect descendents are found, and marked as classes representing persistent data. For example, in a typical J2EE environment, the persistent data is represented by the Entity Beans. So, the set of all Entity Beans represent the persistent data with which the software interacts. The classes that modify the persistent data are then found by looking for non-read-only calls to the Entity Beans. In this way the producer of the data is identifier. The consumer of data is one that makes read-only calls to Entity Beans. Given this producer/consumer relation for the data, the indirect dependencies are found.

Computer hardware and software

Fig. 5 is a schematic representation of a computer system **100** that can be used to implement the diagnostic techniques described herein. The computer system **100** can be

thought of as a programmer's work station. Computer software executes under a suitable operating system installed on the computer system **100** to assist in performing the described techniques. This computer software is programed using any suitable computer programing language, and may be thought of as comprising various software code means for achieving particular steps.

The components of the computer system **100** include a computer **120**, a keyboard **110** and mouse **115**, and a video display **190**. The computer **120** includes a processor **140**, a memory **150**, input/output (I/O) interfaces **160**, **165**, a video interface **145**, and a storage device **155**.

The processor **140** is a central processing unit (CPU) that executes the operating system and the computer software executing under the operating system. The memory **150** includes random access memory (RAM) and read-only memory (ROM), and is used under direction of the processor **140**.

The video interface **145** is connected to video display **190** and provides video signals for display on the video display **190**. User input to operate the computer **120** is provided from the keyboard **110** and mouse **115**. The storage device **155** can include a disk drive or any other suitable storage medium.

Each of the components of the computer **120** is connected to an internal bus **130** that includes data, address, and control buses, to allow components of the computer **120** to communicate with each other via the bus **130**.

The computer system **100** can be connected to one or more other similar computers via a input/output (I/O) interface **165** using a communication channel **185** to a network, represented as the Internet **180**. In this way, a distributed team can co-operate in terms of portions of code being written or hosted from the other locations.

The computer software may be recorded on a portable storage medium, in which case, the computer software program is accessed by the computer system **100** from the storage device **155**. Alternatively, the computer software can be accessed directly from the Internet **180** by the computer **120**. In either case, a user can interact with the computer

system **100** using the keyboard **110** and mouse **115** to operate the programmed computer software executing on the computer **120**.

Other configurations or types of computer systems can be equally well used to implement the described techniques. The computer system **100** described above is described only as an example of a particular type of system suitable for implementing the described techniques.

Conclusion

As a tool, the methodology greatly reduces the debugging effort required to manage the code in a distributed development environment.

Various alterations and modifications can be made to the techniques and arrangements described herein, as would be apparent to one skilled in the relevant art.